MEM 591 – Fall 2022

Homework 3

Khac Hieu Dinh (Personal Submission)

Problem 1: Solving Linear System with LU Decomposition with Partial Pivoting

Solution Approach: The U matrix is obtained the same way as in Gaussian elimination with pivoting (see HW2). For LU decomposition, however, the pseudo-lower-diagonal $\bar{L}$ matrix (lower diagonal, but the rows are shuffled) also needs to be calculated.

$$(M_n P_n \dots M_2 P_2 M_1 P_1)A = U$$

Row operations on A (row swaps and annihilation) applied in order from right to left: P_1, then M_1, then P_2, then M_2 etc.

$$A = (M_n P_n \dots M_2 P_2 M_1 P_1)^{-1} U = (P_1^{-1} M_1^{-1} \dots P_n^{-1} M_n^{-1})U = \bar{L}U$$

$$\bar{L} = I_n (P_1^{-1} M_1^{-1} \dots P_n^{-1} M_n^{-1}) \text{ (successive column operation)}$$

That is:

- To build the U matrix: Apply $P_1 \rightarrow M_1 \rightarrow P_2 \rightarrow M_2$ ... to A as ROW operation (pre-multiplication)
- To build the $\bar{L}$ matrix: Apply $P_1^{-1} \rightarrow M_1^{-1} \rightarrow P_2^{-1} \rightarrow M_2^{-1}$ ... to A as COLUMN operation (post-multiplication)

Note that $P_n^{-1} = P_n$ (to undo the swapping of two rows, swap them back). Therefore, in the program, every time the rows of A are swapped, the columns with same index of $\bar{L}$ are also swapped.

The cumulative result of row swaps/column swaps $P = P_1 P_2 \dots P_n$ (which we will need to make $\bar{L}$ truly lower-triangular) is mathematically tracked by a permutation matrix. Programmatically, however, the row swaps are tracked with an n-by-1 vector. The P vector is initialized to the following state:

$$P = [1\ 2\ 3\ 4\ 5]^T$$

That is: the entry at index n has value n. As the row swapping is applied on A (due to partial pivoting), it is also applied on P. For example, rows 2 and 4 are swapped:

$$P = [1\ 4\ 3\ 2\ 5]$$

, followed by the swapping of the (current) row 2 and 5:

$$P = [1\ 5\ 3\ 2\ 4]$$

If the corresponding columns are swapped (see how $\bar{L}$ is constructed above) in an identity matrix, the result would be the following matrix:

$$\begin{bmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{bmatrix} \rightarrow \begin{bmatrix}1&0&0&0&0\\0&0&0&1&0\\0&0&1&0&0\\0&1&0&0&0\\0&0&0&0&1\end{bmatrix} \rightarrow \begin{bmatrix}1&0&0&0&0\\0&0&0&1&0\\0&0&1&0&0\\0&0&0&0&1\\0&1&0&0&0\end{bmatrix}$$

Looking at the P vector, we see that to find where one entry is on column n, we just have look at the n^{th} entry in the P vector. For example, $P(4) = 2$, so we know that in the 4th column, the 1 is located on the second row.

By knowing exactly where the 1 entry on each column are after (thanks to the P vector), we can now perform the column operations associated with the inverted annihilation matrix M_n^{-1} (which is just M_n without the negative sign before the multiplication factor).

Example:

$$A = \begin{bmatrix}2&4&5\\0&9&1\\0&3&1\end{bmatrix} \qquad M = \begin{bmatrix}1&0&0\\0&1&0\\0&-\frac{1}{3}&1\end{bmatrix} \qquad M^{-1} = \begin{bmatrix}1&0&0\\0&1&0\\0&\frac{1}{3}&1\end{bmatrix}$$

The current state of the permutation vector: $P = [3\ 1\ 2]^T$

To annihilate the entry in red, subtract 1/3 of the second row from the third row. The appropriate COLUMN action to perform on the (current state of) $\bar{L}$ matrix is to add 1/3 of the third column to the second column. HOWEVER, due to pivoting (row swap in A, column swap in $\bar{L}$), we cannot guarantee that $\bar{L}(3,3)$ is 1 and, thus, cannot just place the multiplier of 1/3 at $\bar{L}(3,2)$. To know which entry on the 3 column is 1, we look at $P(3) = 2$. Therefore, the third column is $[0\ 1\ 0]^T$ (the index of the pivot column is always less than the index of the annihilated row, so at any given pivot step number k, the columns to the right of column k in $\bar{L}$ is guaranteed to be "untouched" identity column). Thus, we place the multiplier 1/3 at $\bar{L}(2,2)$.

Once $\bar{L}$ has been constructed, we can use the permutation vector to "unscramble" it into the true lower-diagonal matrix L:

$$PA = P\bar{L}U = LU \quad \text{(see lecture slide)}$$

$$P\bar{L} = L$$

That is: swap the rows of $\bar{L}$ the same way the rows of A were swapped in the construction of U. The P vector contains the (final) record of how the rows of A were swapped.

$$\mathrm{P} = [2\ 3\ 1]^{\mathrm{T}} \rightarrow \text{Put row 2, row 3, row 1 into position 1, 2, and 3, respectively}$$

In MATLAB, this is accomplished simply by non-consecutive indexing:

```
L = L(P,:)
```

Similarly, if the original problem is $Ax = b$, then, to obtain an equivalent problem $PAx = Pb$, the rows of b must also be swapped in the same manner as A was swapped during partial pivoting Gaussian elimination:

```
b = b(P)
```

With the original problem transformed into a true LU decomposition (instead of having to deal with pseudo lower diagonal), the solution of the linear system can be obtained by solving 2 sub-problems:

$$Ly = Pb = b_{shuffled}$$

$$Ux = y$$

, which has been accomplished in previous homework.

Code:

```
function x = myLUPartialPivotLinSolve(A,b)
% myLUDecomp: personal implementation of LU decomposition without pivoting
n = size(A,1);
L = eye(n); % Starting point for L matrix
P = 1:n; % Permutation Tracker Array
zero_tol = 1e-8; % Cannot use == to compare floating point number

for pivot_index = 1:n % Pivoting from column 1 to column n

    % Find the index with maximum pivot in this column, below the current
    % pivot row. LOCAL pivot index in the range of the rows searched
    [~, max_pivot_relative_index] = max(abs(A(pivot_index:end,pivot_index)));
    % Add the current pivot index and subtract 1 to get the global max pivot index
    max_pivot_absolute_index = max_pivot_relative_index + pivot_index - 1;

    % If the current pivot row isn't already the maximum pivot row:
    % Swap the rows in matrix A
    % Swap the entries of P the same way A is swapped (P keep tracks of
    % the index of each original row: P(3) = the index of the
    % original 3rd row after the swappings)
    % Swap the columns in maxtrix L (right multiplication = column
    % operations, inverse of row swap matrix = itself)
    if max_pivot_absolute_index ~= pivot_index

        A([pivot_index, max_pivot_absolute_index],:) =
A([max_pivot_absolute_index,pivot_index],:);
        P([pivot_index, max_pivot_absolute_index]) =
P([max_pivot_absolute_index,pivot_index]);
        L(:,[pivot_index, max_pivot_absolute_index]) =
L(:,[max_pivot_absolute_index,pivot_index]);
    end

    pivot = A(pivot_index,pivot_index);

    % If the max pivot is zero -> zero sub column -> skip to next pivot
    % (problem has no unique solution)
    if abs(pivot) < zero_tol % checking floating point zero
        continue
    end
```

```
    for elim_row_index = (pivot_index+1):n % start eliminating rows below the current
pivot

        multiplier = A(elim_row_index,pivot_index)/pivot; % calculating multiplier

        L(P(elim_row_index), pivot_index) = multiplier; % Interpreting the inverse
elementary operation AS A COLUNM OPERATOR (RIGHT MULTIPLICATION)

        A(elim_row_index,pivot_index) = 0; %% Save 1 calculation, we already know
that this should be zero

        for elim_col_index = (pivot_index+1):n % Subtract multiplier * times pivot
row from the eliminated column
            A(elim_row_index,elim_col_index) = A(elim_row_index,elim_col_index) -
A(pivot_index,elim_col_index)*multiplier;
        end

    end

end

L_debug = L % Debug print pseudo-diagonal L to compare wuth lu()
U_debug = A % Debug print U to compare wuth lu()

% To get L as a lower diagonal matrix, apply the overall row swap to the
% current L. Row swap order encoded in P array
% To get the b that would not change the problem, also apply the
b = b(P);
L = L(P,:) % DEBUG OUTPUT: PL = True lower diagonal

% With L being true Lower Diagonal, the Forward Substitution algorithm
% written in HW 1 can be used
y = myForwardSubstitution(L,b);
x = myBackSubstitution(A,y);

end
```

Results:

$$A = \begin{bmatrix} 3 & 4 & -2 \\ 8 & 5 & -4 \\ -6 & -2 & 3 \end{bmatrix}; \qquad b = \begin{bmatrix} 3 \\ 6 \\ 8 \end{bmatrix}$$

$$Ax = b$$

Comparison Between Personal Implementation of LU Decomposition with Pivoting and MATLAB's lu() (which also implements partial pivoting)

L_debug = 0.3750 1.0000 0 1.0000 0 0 -0.7500 0.8235 1.0000 U_debug = 8.0000 5.0000 -4.0000 0 2.1250 -0.5000 0 0 0.4118	matlab_L = 0.3750 1.0000 0 1.0000 0 0 -0.7500 0.8235 1.0000 matlab_U = 8.0000 5.0000 -4.0000 0 2.1250 -0.5000 0 0 0.4118

True Lower Diagonal Decomposition by Applying the Row Permutation on $\bar{L}$

```
L = L(P,:) // See implementation above, output unsuppressed
```

```
L =

    1.0000         0         0
    0.3750    1.0000         0
   -0.7500    0.8235    1.0000
```

(true lower diagonal)

Linear System Solution Result Verification with A\b

my_x = 10.7143 7.1429 28.8571	matlab_x = 10.7143 7.1429 28.8571

Problem 2: Implement Polynomial Regression (of Arbitrary Order)

In general, to curve fit with a linear combination of m basis function $f_m(t)$ to n data points (t_n, y_n), simply create the A matrix which has $f_m(t_n)$ as the entry at row n, column m. The curve fit problem is then transformed into the least square regression problem:

$$Ax = y$$

, where x is the vector containing the coefficients of the basis function that would result in the least P2 norm of the error.

In this case, the basis functions are polynomial terms: $f_m(t) = t^m$

Structure-wise, the curve-fit subroutine consists of 2 parts: the A matrix generation and the least-square solver. For this problem, the algorithm for finding the solution for the least square problem is solving the normal equation $A^T Ax = A^T y$ (using Cholesky decomposition), although the QR decomposition method can be easily implemented by swapping out the least-square solver (see Appendix A for the version of polynomial curve fitting that uses QR decomposition).

Implementation of Least Square Solver (Normal Equation Method)

```
function x = myLinRegressNormalEqn(A,b)

normalA = myMatrixMatrixMult(transpose(A),A);
normalb = myMatrixMult(transpose(A),b);

x = myLinearSolveChol(normalA,normalb); // see HW2

end
```

Implementation of Polynomial Curve Fitting

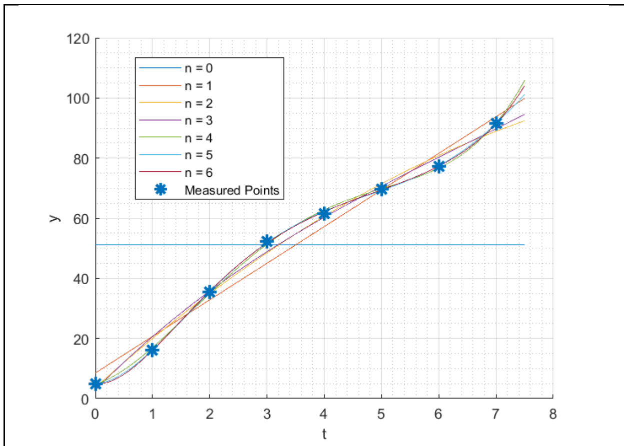
```
function coeffs = myPolyFitNormalEqn(independent, dependent, deg)

num_row = length(independent);

if num_row ~= length(dependent)
    error("Array Size Mismatch")
end

poly_lambda = @(x, n) x^n;


A = zeros(length(independent),deg+1);

for col = 0:deg
    for row = 1:num_row
        A(row,col+1) = poly_lambda(independent(row),col);
    end
end

coeffs = myLinRegressNormalEqn(A,dependent);

end
```

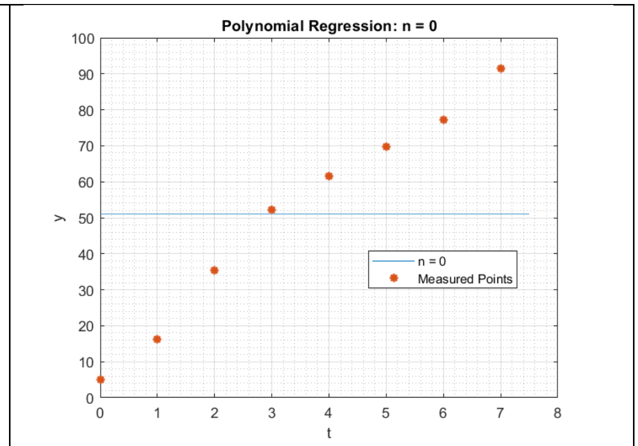
Results

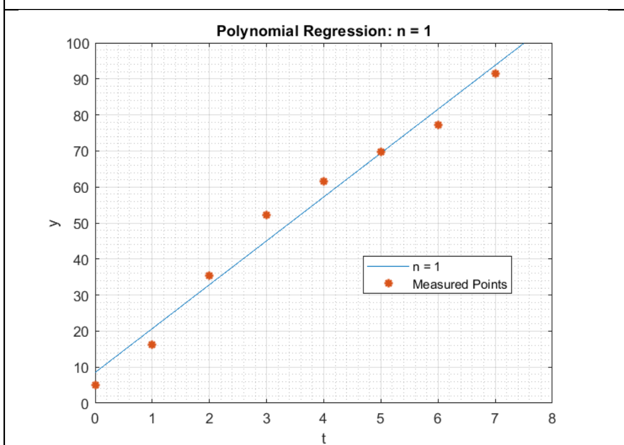
Coefficient Tables:

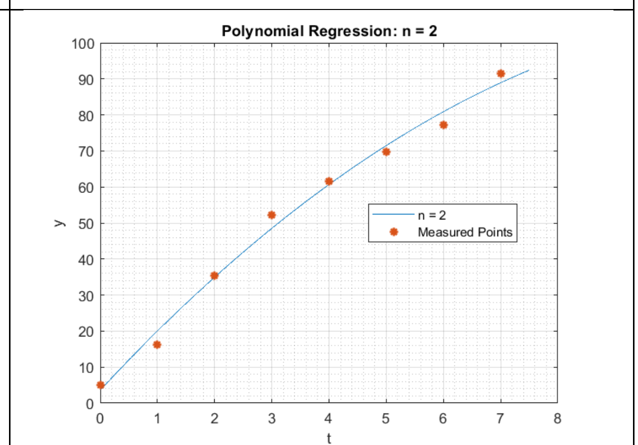
Order	Coefficients (Left to Right : from Low Order to High Order)
0	51.1250
1	8.5000 12.1786
2	3.6583 17.0202 -0.6917
3	2.8152 19.2286 -1.5348 0.0803
4	4.7561 5.9657 8.2852 -2.1841 0.1617
5	4.9907 0.9170 14.4439 -4.6867 0.5723 -0.0235
6	5.0164 -1.7335 19.0927 -7.5264 1.3536 -0.1226 0.0047

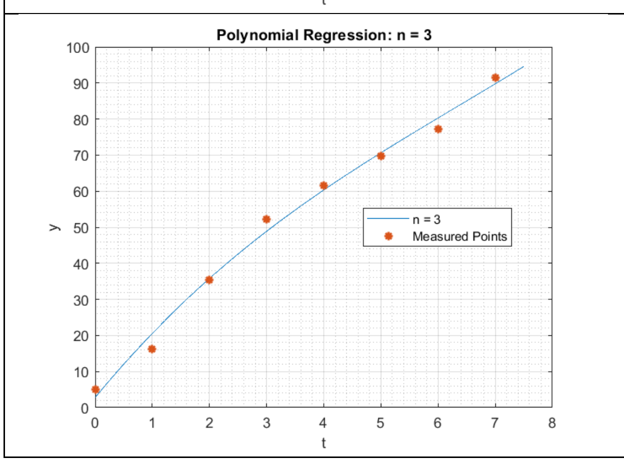
Plots:

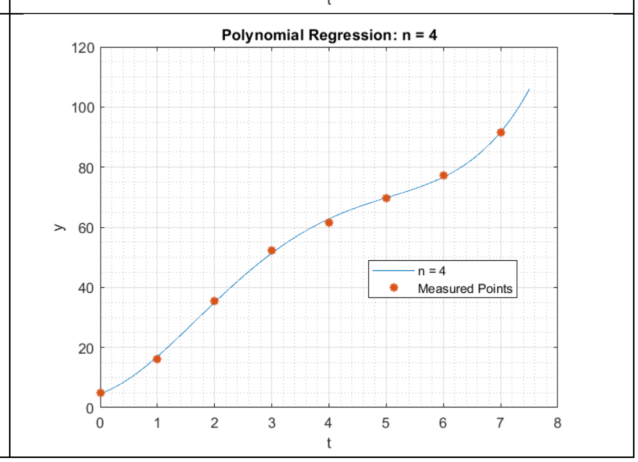

n = 0
n = 1
n = 2
n = 3
n = 4
n = 5
n = 6
Measured Points
y
t

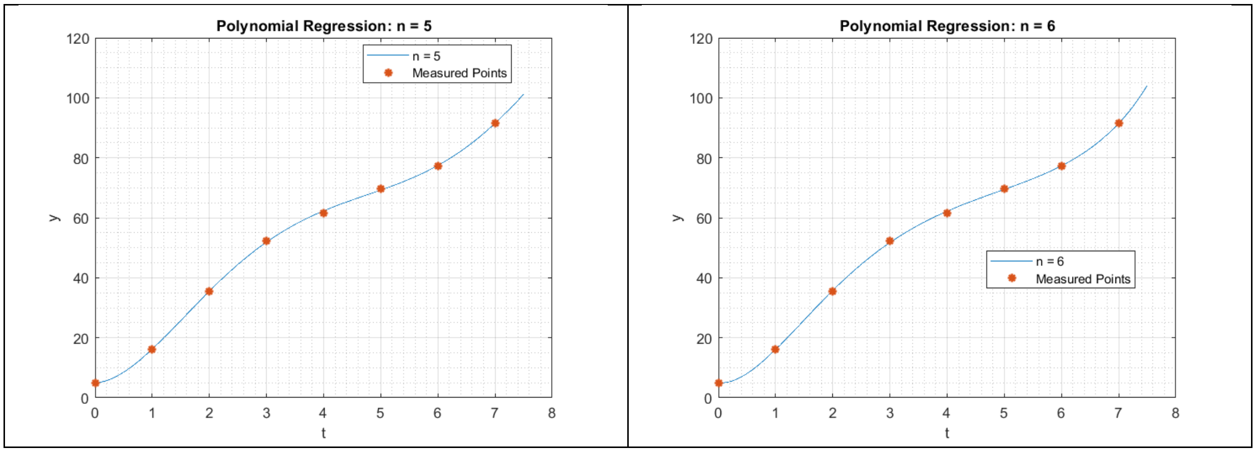

Polynomial Regression: n = 0
n = 0
Measured Points
y
t


Polynomial Regression: n = 1
n = 1
Measured Points
y
t


Polynomial Regression: n = 2
n = 2
Measured Points
y
t


Polynomial Regression: n = 3
n = 3
Measured Points
y
t


Polynomial Regression: n = 4
n = 4
Measured Points
y
t



Problem 3: Householder QR Factorization

(The following calculations was carried out manually)

$$A = \begin{bmatrix} 0.26 & 0.15 \\ 0.29 & 0.31 \\ 2.05 & 1.49 \end{bmatrix}; \qquad b = \begin{bmatrix} 0.56 \\ 0.78 \\ 2.24 \end{bmatrix}$$

Step 1: First pivot annihilation (H_1 transformation):

$$\boldsymbol{a_1} = \begin{bmatrix} 0.26 \\ 0.29 \\ 2.05 \end{bmatrix}; \qquad \alpha_1 = -\|\boldsymbol{a_1}\|_2 = -\sqrt{0.26^2 + 0.29^2 + 2.05^2} = -2.0867$$

(The negative sign was used because the diagonal entry, 0.26, is positive, and we do not want cancellation of the first entry).

$$\boldsymbol{v_1} = \boldsymbol{a_1} - \alpha_1 \boldsymbol{e_1} = \begin{bmatrix} 0.26 \\ 0.29 \\ 2.05 \end{bmatrix} - \begin{bmatrix} -2.0867 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2.3467 \\ 0.2900 \\ 2.0500 \end{bmatrix}$$

Verifying that the transformation works (i.e., the entries below $\boldsymbol{A}(\mathbf{1},\mathbf{1})$ should be annihilated, and $\boldsymbol{A}(\mathbf{1},\mathbf{1})$ should become $\boldsymbol{\alpha_1}$):

$$\boldsymbol{H_1}(\boldsymbol{a_1}) = \boldsymbol{a_1} - 2\frac{\boldsymbol{v_1^T a_1}}{\boldsymbol{v_1^T v_1}}\boldsymbol{v_1} = \begin{bmatrix} 0.26 \\ 0.29 \\ 2.05 \end{bmatrix} - 2\frac{(2.3467 \times 0.26 + 0.2900^2 + 2.0500^2)}{(2.3467^2 + 0.2900^2 + 2.0500^2)}\begin{bmatrix} 2.3467 \\ 0.2900 \\ 2.0500 \end{bmatrix}$$

$$= \begin{bmatrix} 0.26 \\ 0.29 \\ 2.05 \end{bmatrix} - 2 \times \frac{4.8967}{9.7935} \times \begin{bmatrix} 2.3467 \\ 0.2900 \\ 2.0500 \end{bmatrix} = \begin{bmatrix} 0.26 \\ 0.29 \\ 2.05 \end{bmatrix} - 1 \times \begin{bmatrix} 2.3467 \\ 0.2900 \\ 2.0500 \end{bmatrix} = \begin{bmatrix} -2.0867 \\ 0 \\ 0 \end{bmatrix}$$

Applying the transformation on the second column of A:

$$\boldsymbol{H_1}\left(\begin{bmatrix} 0.15 \\ 0.31 \\ 1.49 \end{bmatrix}\right) = \begin{bmatrix} 0.15 \\ 0.31 \\ 1.49 \end{bmatrix} - 2\frac{2.3467 \times 0.15 + 0.2900 \times 0.31 + 2.0500 \times 1.49}{(2.3467^2 + 0.2900^2 + 2.0500^2)}\begin{bmatrix} 2.3467 \\ 0.2900 \\ 2.0500 \end{bmatrix}$$

$$= \begin{bmatrix} 0.15 \\ 0.31 \\ 1.49 \end{bmatrix} - 2 \times \frac{3.4964}{9.7935} \times \begin{bmatrix} 2.3467 \\ 0.2900 \\ 2.0500 \end{bmatrix} = \begin{bmatrix} -1.5256 \\ 0.1029 \\ 0.0262 \end{bmatrix}$$

Current state of A matrix:

$$H_1 A = \begin{bmatrix} -2.0867 & -1.5256 \\ 0 & 0.1029 \\ 0 & 0.0262 \end{bmatrix}$$

Applying the transformation on the non-homogenous vector b:

$$H_1(b) = \begin{bmatrix} 0.56 \\ 0.78 \\ 2.24 \end{bmatrix} - 2 \times \frac{2.3467 \times 0.56 + 0.2900 \times 0.78 + 2.0500 \times 2.24}{(2.3467^2 + 0.2900^2 + 2.0500^2)} \times \begin{bmatrix} 2.3467 \\ 0.2900 \\ 2.0500 \end{bmatrix}$$

$$= \begin{bmatrix} 0.56 \\ 0.78 \\ 2.24 \end{bmatrix} - 2 \times \frac{6.1323}{9.7935} \times \begin{bmatrix} 2.3467 \\ 0.2900 \\ 2.0500 \end{bmatrix} = \begin{bmatrix} -2.3788 \\ 0.4168 \\ -0.3273 \end{bmatrix}$$

Step 2: Second pivot annihilation (H_2 transformation):

$$\boldsymbol{a_2} = \begin{bmatrix} 0 \\ 0.1029 \\ 0.0262 \end{bmatrix}; \qquad \alpha_2 = -\|\boldsymbol{a_2}\|_2 = -\sqrt{0.1029^2 + 0.0262^2} = -0.1062$$

$$\boldsymbol{v_2} = \boldsymbol{a_2} - \alpha_2 \boldsymbol{e_2} = \begin{bmatrix} 0 \\ 0.1029 \\ 0.0262 \end{bmatrix} - \begin{bmatrix} 0 \\ -0.1062 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.2092 \\ 0.0262 \end{bmatrix}$$

Verification (application to the second column of $\boldsymbol{H_1 A}$)

$$H_2\left(\begin{bmatrix} -1.5256 \\ 0.1029 \\ 0.0262 \end{bmatrix}\right) = \begin{bmatrix} -1.5256 \\ 0.1029 \\ 0.0262 \end{bmatrix} - 2 \times \frac{0.2092 \times 0.1029 + 0.0262 \times 0.0262}{(0.2092^2 + 0.0262^2)} \times \begin{bmatrix} 0 \\ 0.2092 \\ 0.0262 \end{bmatrix}$$

$$= \begin{bmatrix} -1.5256 \\ -0.1062 \\ 0 \end{bmatrix} \ (ok)$$

(Note that entries above the diagonal are untouched)

Applying the transformation on the non-homogenous vector $H_1 b$:

$$H_2(H_1 b) = \begin{bmatrix} -2.3788 \\ 0.4168 \\ -0.3273 \end{bmatrix} - 2 \times \frac{0.2092 \times 0.4168 + 0.0262 \times -0.3273}{(0.2092^2 + 0.0262^2)} \times \begin{bmatrix} 0 \\ 0.2092 \\ 0.0262 \end{bmatrix}$$

$$= \begin{bmatrix} -2.3788 \\ -0.3230 \\ -0.4201 \end{bmatrix}$$

Equivalent System of Linear Equation

$$H_2 H_1 A x = H_2 H_1 b$$

$$\begin{bmatrix} R \\ 0 \end{bmatrix} x = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$\begin{bmatrix} -2.0867 & -1.5256 \\ 0 & -0.1062 \\ 0 & 0 \end{bmatrix} x = \begin{bmatrix} -2.3788 \\ -0.3230 \\ -0.4201 \end{bmatrix}$$

Eliminating the third row to obtain a back-substitution problem:

$$Rx = b_1$$

$$\begin{bmatrix} -2.0867 & -1.5256 \\ 0 & -0.1062 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2.3788 \\ -0.3230 \end{bmatrix}$$

Step 3: Back-substitution

$$\boldsymbol{x_2} = \frac{-0.3230}{-0.1062} = \mathbf{3.0411}$$

$$(-2.0867)\boldsymbol{x_1} + (-1.5256)\boldsymbol{x_2} = -2.3788$$

$$\rightarrow \boldsymbol{x_1} = \frac{-2.3788 - (-1.5256)3.0411}{(-2.0867)} = \mathbf{-1.0834}$$

$$\boldsymbol{x} = \begin{bmatrix} \mathbf{-1.0834} \\ \mathbf{3.0411} \end{bmatrix}$$

Verification with A/b

```
A\b
```

```
ans =

   -1.0834
    3.0411
```

Problem 4: Application of Householder QR Factorization in Linear Regression

The corresponding least-square system $Ax = b$ is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1.95 \\ 2.74 \\ -2.45 \\ 3.32 \\ 1.23 \\ 4.45 \\ 1.61 \\ 3.21 \\ 0.45 \\ -2.75 \end{bmatrix}$$

The Householder QR factorization subroutine is identical to the approach used in problem 3: for each pivot, apply the annihilation transformation to the columns after the pivot columns and b vector. Once the R diagonal matrix has been obtained, use back-substitution to find the solution for the least-square problem.

Subroutine:

```
function x = myLinRegressHouseholder(A,b)


numCol = size(A,2);

for pivot_index = 1:numCol

    aSubCol = A(pivot_index:end,pivot_index);
    vSubCol = aSubCol;
    alpha = -my2Norm(aSubCol)*sign(aSubCol(1));
    vSubCol(1) = aSubCol(1) - alpha;
    A(pivot_index,pivot_index) = alpha;
    A(pivot_index+1:end,pivot_index) = 0;

    for transformIndex = (pivot_index + 1):numCol
        transformSubCol = A(pivot_index:end,transformIndex);
        A(pivot_index:end,transformIndex) = transformSubCol -
2*myInnerProduct(vSubCol,transformSubCol)/myInnerProduct(vSubCol,vSubCol)*vSubCol;
    end
    bSubCol = b(pivot_index:end);
    b(pivot_index:end) = bSubCol -
2*myInnerProduct(vSubCol,bSubCol)/myInnerProduct(vSubCol,vSubCol)*vSubCol;

end

R = A(1:numCol,:);
QTb = b(1:numCol);

x = myBackSubstitution(R,QTb);

end
```

Solution & Comparison with MATLAB

x = myLinRegressHouseholder(A,b)	x_MATLAB = A\b
x = 2.9600 2.1460 -1.4600 1.9140	x_MATLAB = 2.9600 2.1460 -1.4600 1.9140

We can see that the values altitudes that would result in the least P2 norm of the residual is **not the same as the direct measurement.** In fact, the discrepancy is quite large. This is due to that fact that the distances between the points disagree with the direct measurement. For example: $x_1 = 1.95; x_2 = 2.74; x_1 - x_2 \neq 1.23$.

If we take the least-square values as the true value, the percent discrepancy between the direct-measurement and the least square solution is as follows:

```
measured = [1.95 2.74 -2.45 3.32]';

percent_discrepancy = (x-measured)./measured*100
```

```
percent_discrepancy =

   51.7949
  -21.6788
  -40.4082
  -42.3494
```

For x_1, the discrepancy reaches 50%. This raises a lot of questions regarding the accuracy of our measurements.

Problem 5: Knowledge Check

There are always solutions for a linear least squares problem: True

The solution is unique if and only if columns of A are linearly dependent: False

Reason: The solution is unique if and only if columns are linearly INDEPENDENT

If rank(A) < n then A is rank-deficient, and the solution is unique: False

Reason: Rank deficient A -> linearly dependent columns -> No unique Solution!

Multiplying both side of an LLS problem by an orthogonal matrix doesn’t change its solution: True

Appendix

Problem 1 Script

```
clc; clear; close all
A = [3 4 -2;
     8 5 -4;
     -6 -2 3];
b =[3 6 8]';
my_x = myLUPartialPivotLinSolve(A,b)
```

```
L_debug =

    0.3750    1.0000         0
    1.0000         0         0
   -0.7500    0.8235    1.0000


U_debug =

    8.0000    5.0000   -4.0000
         0    2.1250   -0.5000
         0         0    0.4118


L =

    1.0000         0         0
    0.3750    1.0000         0
   -0.7500    0.8235    1.0000


my_x =

   10.7143
    7.1429
   28.8571
```

```
matlab_x = A\b
```

```
matlab_x =

   10.7143
    7.1429
   28.8571
```

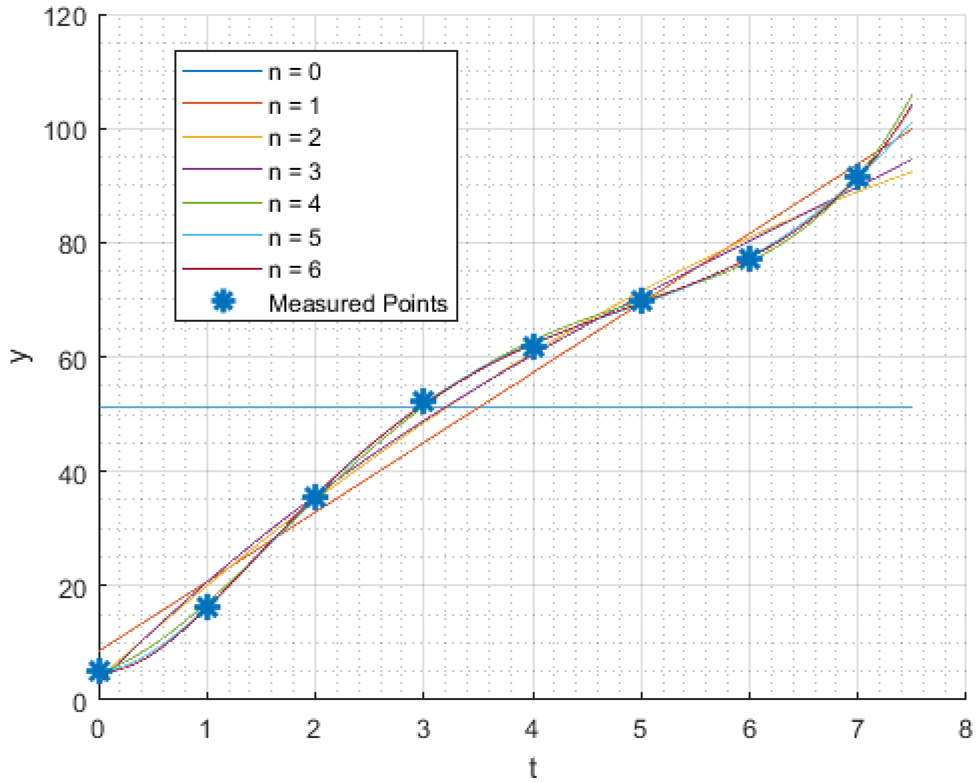
```
[matlab_L, matlab_U] = lu(A)
```

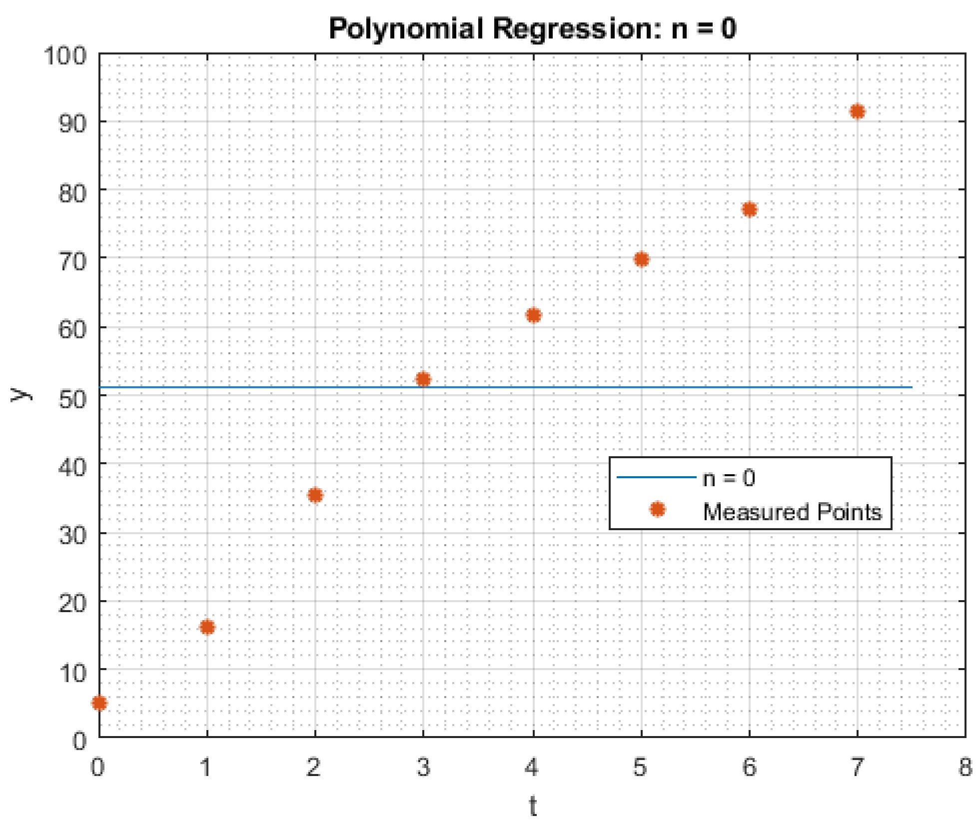
```
matlab_L =

    0.3750    1.0000         0
    1.0000         0         0
   -0.7500    0.8235    1.0000


matlab_U =

    8.0000    5.0000   -4.0000
         0    2.1250   -0.5000
```

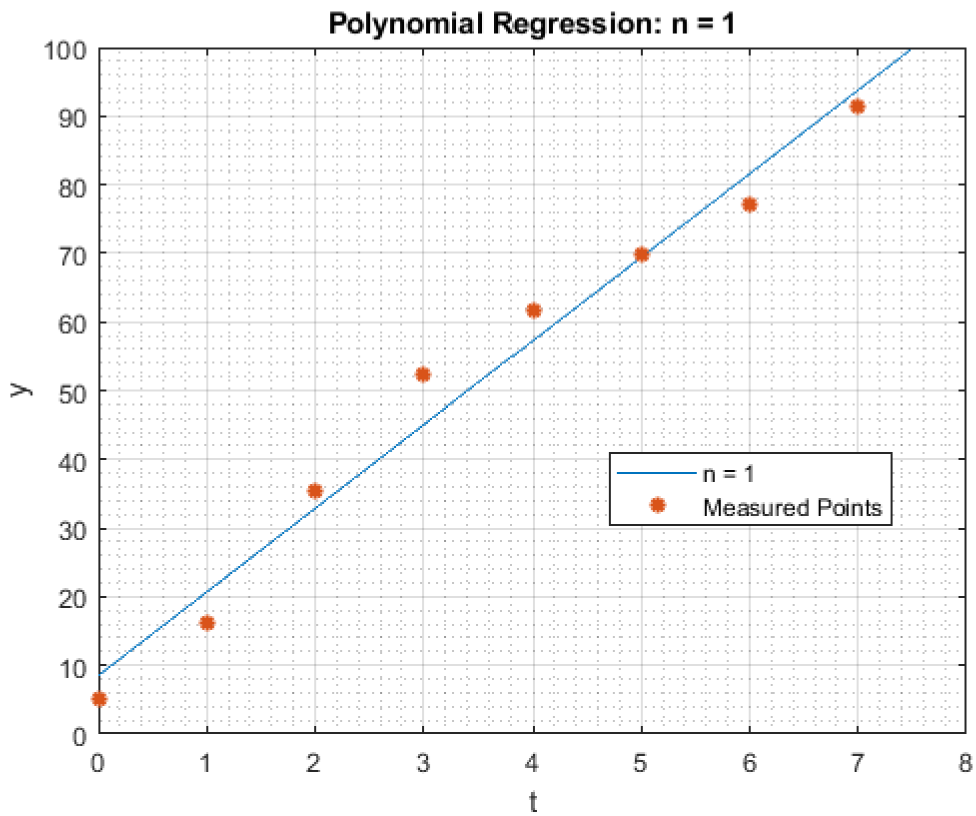
```
         0         0    0.4118
```

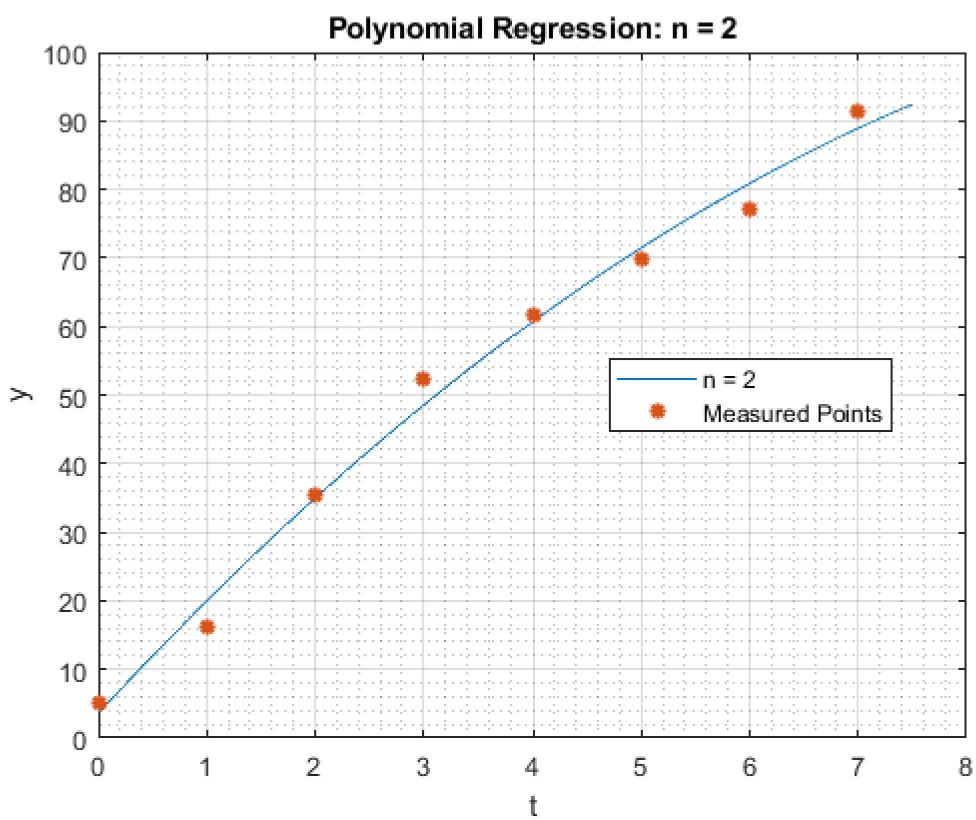
Problem 2 Script

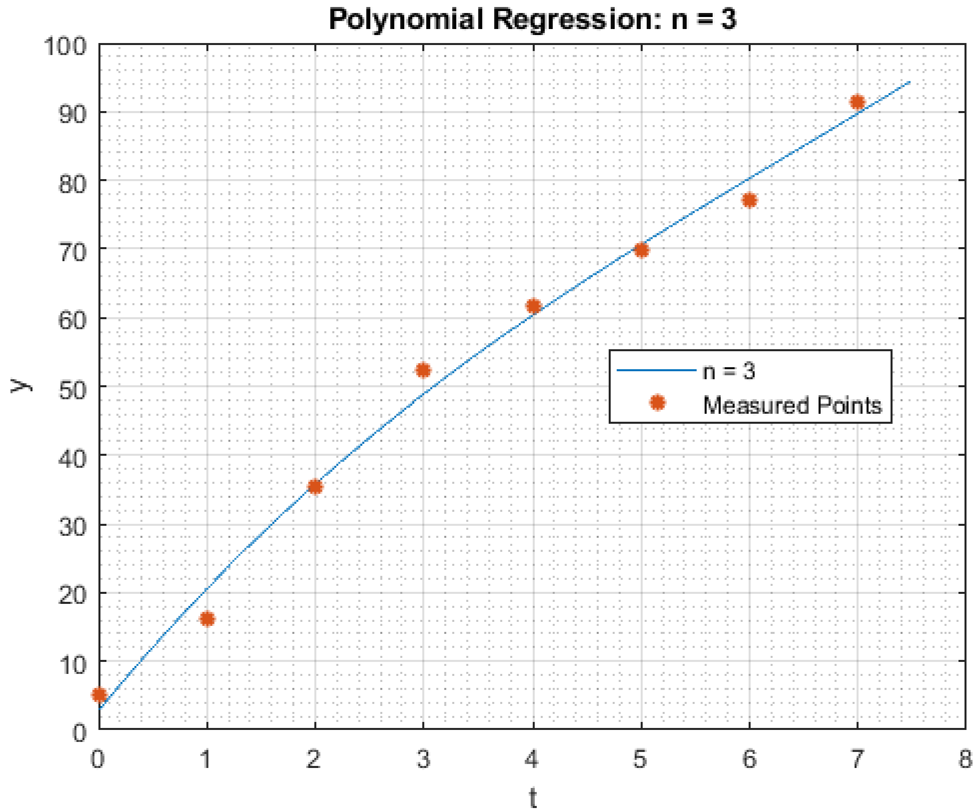
```
clc; clear; close all;
t = 0:7;
b = [5.0 16.2 35.4 52.3 61.6 69.8 77.2 91.5];
max_fit_order = 6;

plot_time = linspace(0,7.5,1000);

figure;
hold on;
for fit_order = 0:max_fit_order
    coeffs = myPolyFitNormalEqn(t,b,fit_order);
    plot_data = zeros(1,length(plot_time));
    for order = 0:fit_order
        plot_data = plot_data + coeffs(order+1).*plot_time.^order;
    end
    plot(plot_time, plot_data,"DisplayName",sprintf("n = %d",fit_order));
end
xlabel("t")
ylabel("y")
plot(t,b,"*","DisplayName","Measured Points",'MarkerSize',10,'Linewidth',2)
hold off;
grid on; grid minor; legend(Location="best")

for fit_order = 0:max_fit_order
    figure;
    coeffs = myPolyFitNormalEqn(t,b,fit_order);
    plot_data = zeros(1,length(plot_time));
    for order = 0:fit_order
        plot_data = plot_data + coeffs(order+1).*plot_time.^order;
    end
    plot(plot_time, plot_data,"DisplayName",sprintf("n = %d",fit_order)); hold on;
    plot(t,b,"*","DisplayName","Measured Points",'Linewidth',1.5); hold off;
    title(sprintf("Polynomial Regression: n = %d",fit_order))
    xlabel("t")
    ylabel("y")
    grid on; grid minor; legend(Location="best")
end
```


n = 0
n = 1
n = 2
n = 3
n = 4
n = 5
n = 6
Measured Points
y
t

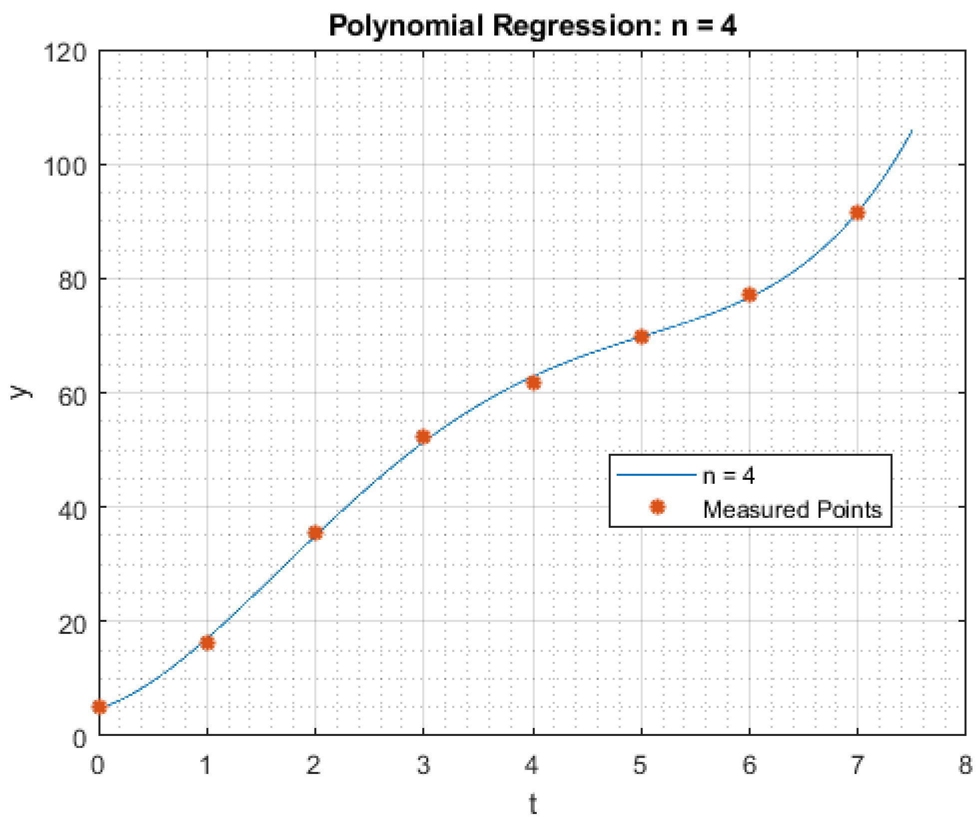

Polynomial Regression: n = 0
n = 0
Measured Points
y
t


Polynomial Regression: n = 1
100
90
80
70
60
50
40
30
20
10
0
y
0
1
2
3
4
5
6
7
8
t
n = 1
Measured Points

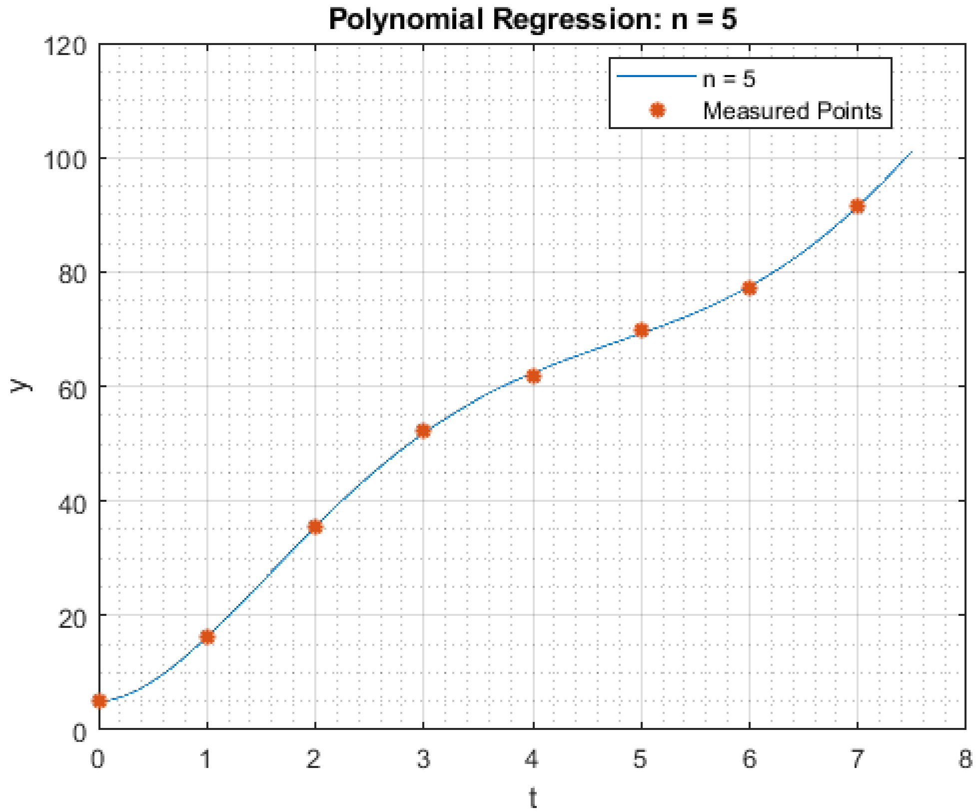

Polynomial Regression: n = 2
100
90
80
70
60
50
40
30
20
10
0
y
0
1
2
3
4
5
6
7
8
t
n = 2
Measured Points

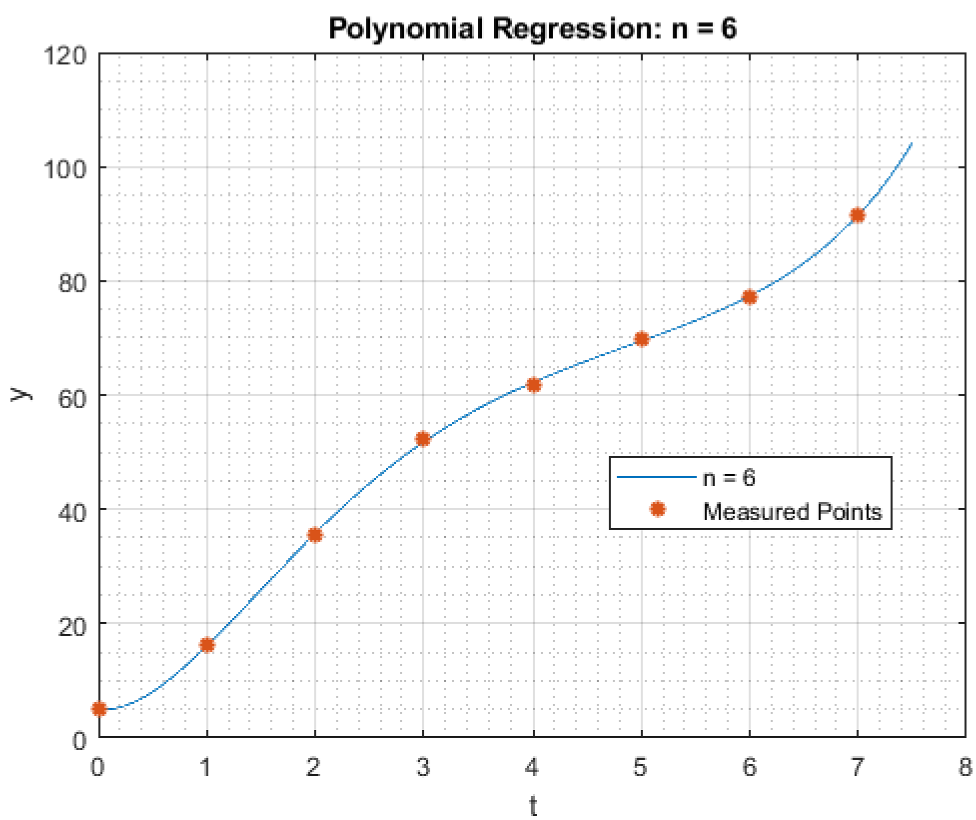


Polynomial Regression: n = 3
100
90
80
70
60
50
40
30
20
10
0
y
0
1
2
3
4
5
6
7
8
t
n = 3
Measured Points



Polynomial Regression: n = 4
120
100
80
60
40
20
0
y
0
1
2
3
4
5
6
7
8
t
n = 4
Measured Points


Polynomial Regression: n = 5
n = 5
Measured Points
y
t
0
20
40
60
80
100
120
0
1
2
3
4
5
6
7
8


Polynomial Regression: n = 6
n = 6
Measured Points
y
t
0
20
40
60
80
100
120
0
1
2
3
4
5
6
7
8

Problem 3 Script

Contents


```
A = [
0.26 0.15;
0.29 0.31;
2.05 1.49;
];

b = [0.56 0.78 2.24]';
```

Step 1

```
a1 = A(:,1)
alpha1 = -my2Norm(a1)
v1 = a1 - alpha1*[1 0 0]'
num11 = myInnerProduct(v1,a1)
den11 = myInnerProduct(v1,v1)
H1a1 = a1 - 2*num11/den11*v1

col2 = A(:,2);
num12 = myInnerProduct(v1,col2)
den12 = den11;
H1col2 = col2 - 2*num12/den12*v1

num1b = myInnerProduct(v1,b)
den1b = den11;
H1b = b-2*num1b/den1b*v1
```

```
a1 =

    0.2600
    0.2900
    2.0500


alpha1 =

   -2.0867


v1 =

    2.3467
    0.2900
    2.0500


num11 =

    4.8967


den11 =

    9.7935
```

```
H1a1 =

   -2.0867
   -0.0000
   -0.0000


num12 =

    3.4964


H1col2 =

   -1.5256
    0.1029
    0.0262


num1b =

    6.1323


H1b =

   -2.3788
    0.4168
   -0.3273
```

Step 2

```
a2 = [0; H1col2(2:end)]
alpha2 = -my2Norm(a2)
v2 = a2 - alpha2*[0 1 0]'
num2 = myInnerProduct(v2,H1col2)
den2 = myInnerProduct(v2,v2)
H2col2 = H1col2 - 2*num2/den2*v2

num2b = myInnerProduct(v2,H1b)
den2b = myInnerProduct(v2,v2)
H2H1b = H1b - 2*num2b/den2b*v2
```

```
a2 =

         0
    0.1029
    0.0262


alpha2 =

   -0.1062


v2 =

         0
    0.2092
    0.0262
```

```
num2 =

    0.0222


den2 =

    0.0444


H2col2 =

   -1.5256
   -0.1062
    0.0000


num2b =

    0.0786


den2b =

    0.0444


H2H1b =

   -2.3788
   -0.3230
   -0.4201
```

Step 3

```
R = [[alpha1 0]', H2col2(1:2)]
b1 = H2H1b(1:2)

x2 = b1(2)/R(2,2)
x1 = (b1(1) - x2*R(1,2))/R(1,1)
```

```
R =

   -2.0867   -1.5256
         0   -0.1062


b1 =

   -2.3788
   -0.3230


x2 =

    3.0411


x1 =

   -1.0834
```

Verification

```
A\b
```

```
ans =

   -1.0834
    3.0411
```

Problem 4 Script

```
A = [
eye(4);
1 -1 0 0;
1 0 -1 0;
1 0 0 -1;
0 1 -1 0;
0 1 0 -1;
0 0 1 -1
];
b = [1.95 2.74 -2.45 3.32 1.23 4.45 1.61 3.21 0.45 -2.75]';
x = myLinRegressHouseholder(A,b)
```

```
x =

    2.9600
    2.1460
   -1.4600
    1.9140
```

```
x_MATLAB = A\b
```

```
x_MATLAB =

    2.9600
    2.1460
   -1.4600
    1.9140
```

```
measured = [1.95 2.74 -2.45 3.32]';
percent_discrepancy = (x-measured)./measured*100
```

```
percent_discrepancy =

   51.7949
  -21.6788
  -40.4082
  -42.3494
```

New Function Definitions

```
function x = myLUPartialPivotLinSolve(A,b)
% myLUDecomp: personal implementation of LU decomposition without pivoting
n = size(A,1);
L = eye(n); % Starting point for L matrix
P = 1:n; % Permutation Tracker Array
zero_tol = 1e-8; % Cannot use == to compare floating point number
```

```
for pivot_index = 1:n % Pivoting from column 1 to column n

    % Find the index with maximum pivot in this column, below the current
    % pivot row. LOCAL pivot index in the range of the rows searched
    [~, max_pivot_relative_index] = max(abs(A(pivot_index:end,pivot_index)));
    % Add the current pivot index and subtract 1 to get the global max pivot index
    max_pivot_absolute_index = max_pivot_relative_index + pivot_index - 1;

    % If the current pivot row isn't already the maximum pivot row:
    % Swap the rows in matrix A
    % Swap the entries of P the same way A is swapped (P keep tracks of
    % the index of each original row: P(3) = the index of the
    % original 3rd row after the swappings)
    % Swap the columns in maxtrix L (right multiplication = column
    % operations, inverse of row swap matrix = itself)
    if max_pivot_absolute_index ~= pivot_index

        A([pivot_index, max_pivot_absolute_index],:) =
A([max_pivot_absolute_index,pivot_index],:);
        P([pivot_index, max_pivot_absolute_index]) =
P([max_pivot_absolute_index,pivot_index]);
        L(:,[pivot_index, max_pivot_absolute_index]) =
L(:,[max_pivot_absolute_index,pivot_index]);
    end

    pivot = A(pivot_index,pivot_index);

    % If the max pivot is zero -> zero sub column -> skip to next pivot
    % (problem has no unique solution)
    if abs(pivot) < zero_tol % checking floating point zero
        continue
    end

    for elim_row_index = (pivot_index+1):n % start eliminating rows below the current
pivot

        multiplier = A(elim_row_index,pivot_index)/pivot; % calculating multiplier

        L(P(elim_row_index), pivot_index) = multiplier; % Interpreting the inverse
elementary operation AS A COLUNM OPERATOR (RIGHT MULTIPLICATION)

        A(elim_row_index,pivot_index) = 0; %% Save 1 calculation, we already know
that this should be zero

        for elim_col_index = (pivot_index+1):n % Subtract multiplier * times pivot
row from the eliminated column
            A(elim_row_index,elim_col_index) = A(elim_row_index,elim_col_index) -
A(pivot_index,elim_col_index)*multiplier;
        end

    end

end
```

```
L_debug = L % Debug print pseudo-diagonal L to compare wuth lu()
U_debug = A % Debug print U to compare wuth lu()

% To get L as a lower diagonal matrix, apply the overall row swap to the
% current L. Row swap order encoded in P array
% To get the b that would not change the problem, also apply the
b = b(P);
L = L(P,:) % DEBUG OUTPUT: PL = True lower diagonal

% With L being true Lower Diagonal, the Forward Substitution algorithm
% written in HW 1 can be used
y = myForwardSubstitution(L,b);
x = myBackSubstitution(A,y);


end
```

```
function coeffs = myPolyFitNormalEqn(independent, dependent, deg)

num_row = length(independent);

if num_row ~= length(dependent)
    error("Array Size Mismatch")
end

poly_lambda = @(x, n) x^n;


A = zeros(length(independent),deg+1);

for col = 0:deg
    for row = 1:num_row
        A(row,col+1) = poly_lambda(independent(row),col);
    end
end

coeffs = myLinRegressNormalEqn(A,dependent);

end
```

```
function x = myLinRegressNormalEqn(A,b)

normalA = myMatrixMatrixMult(transpose(A),A); % General Matrix Multiplication
normalb = myMatrixMult(transpose(A),b); % Matrix Vector Multiplication

x = myLinearSolveChol(normalA,normalb);

end
```

```
function x = myLinRegressHouseholder(A,b)

numCol = size(A,2);
```

```
for pivot_index = 1:numCol

    aSubCol = A(pivot_index:end,pivot_index);
    vSubCol = aSubCol;
    alpha = -my2Norm(aSubCol)*sign(aSubCol(1));
    vSubCol(1) = aSubCol(1) - alpha;
    A(pivot_index,pivot_index) = alpha;
    A(pivot_index+1:end,pivot_index) = 0;

    for transformIndex = (pivot_index + 1):numCol
        transformSubCol = A(pivot_index:end,transformIndex);
        A(pivot_index:end,transformIndex) = transformSubCol -
2*myInnerProduct(vSubCol,transformSubCol)/myInnerProduct(vSubCol,vSubCol)*vSubCol;
    end
    bSubCol = b(pivot_index:end);
    b(pivot_index:end) = bSubCol -
2*myInnerProduct(vSubCol,bSubCol)/myInnerProduct(vSubCol,vSubCol)*vSubCol;

end

R = A(1:numCol,:);
QTb = b(1:numCol);

x = myBackSubstitution(R,QTb);

end
```

Auxiliary Functions

```
function squared_norm = my2Norm(a)
% my2Norm: dot a vector with itself and take the square root
    squared_norm = sqrt(myInnerProduct(a,a));
end
```

```
function [maxVal, maxIndex] = myAbsMax(number_array)
% Given a 1D array, find the number with the maximum magnitude and its
% index inside the array
maxVal = -1;
maxIndex = -1;
for numberIndex = 1:length(number_array)
    current = abs(number_array(numberIndex));
    if current > maxVal
        maxVal = current;
        maxIndex = numberIndex;
    end
end
```

```
function inner_product = myInnerProduct(a,b)
% myInnerProduct: Calculate the inner product between two 1D arrays
    n = length(a);
    inner_product = 0;
```

```
    for index = 1:n
        inner_product = inner_product + a(index)*b(index);
    end
end
```

```
function x = myBackSubstitution(A_upper,b)
% myBackSubstitution: personal implementation of back-substitution

    n = length(b); % Calculate Working Dimension
    x = zeros(n,1); % Allocation for Result Vector
    for subStep = n:-1:1 % Reverse Indexing From n to 1
        % Residual = Dot product of sub-vector on the right of the diagonal
        % entry and the sub-vector of known x entries
        residual = myInnerProduct(A_upper(subStep,subStep+1:n),x(subStep+1:n));
        % x at the current row = b at the current row - residual, then
        % divided by diagonal entry
        x(subStep) = (b(subStep)-residual)/A_upper(subStep,subStep);
    end
end
```

```
function x = myForwardSubstitution(A_lower,b)
% myForwardSubstitution: personal implementation of forward substitution

    n = length(b); % Calculate Working Dimension
    x = zeros(n,1); % Allocation for Result Vector
    for subStep = 1:n % Forward Indexing From 1 to n
        % Residual = Dot product of sub-vector on the left of the diagonal
        % entry and the sub-vector of known x entries
        residual = myInnerProduct(A_lower(subStep,1:subStep-1),x(1:subStep-1));
        % x at the current row = b at the current row - residual, then
        % divided by diagonal entry
        x(subStep) = (b(subStep)-residual)/A_lower(subStep,subStep);
    end
end
```

```
function A = myCholesky(A)
% My implementation of Cholesky factorization
% Algorithm from Lecture Slide and Textbook
n = size(A,1);

for j = 1:n
    for k = 1:(j-1)
        for i = j:n
            A(i,j) = A(i,j) - A(i,k)*A(j,k);
        end
        A(k,j) = 0; % Extra bit to wipe the upper diagonal entries
    end
    A(j,j) = A(j,j)^0.5;

    for k = (j+1):n
        A(k,j) = A(k,j)/A(j,j);
    end
```

```
end

end
```

```
function x = myLinearSolveChol(A,b)
% Solving Linear System Using Chol Decomposition
L = myCholesky(A);

L_T = myTranspose(L);

y = myForwardSubstitution(L,b);
x = myBackSubstitution(L_T,y);
end
```

```
function matProd = myMatrixMatrixMult(matA,matB)

if (size(matA,2) ~= size(matB,1))
    error("Dimension not Compattible")
end

numRow = size(matA,1);
numCol = size(matB,2);

matProd = zeros(numRow,numCol);

for rowIndex = 1:numRow
    for colIndex = 1:numCol
    matProd(rowIndex,colIndex) = myInnerProduct(matA(rowIndex,:),matB(:,colIndex));
    end
end

end
```

```
function mat_mult = myMatrixMult(A,b)
% myMatrixMult: matrix multiplication implementation using column
% perspective
    numRow = size(A,1);
    numCol = size(A,2);
    if numCol ~= length(b)
        error("Dimensional Mismatch")
    end
    mat_mult = zeros(numRow,1);
    for index = 1:numRow
        mat_mult(index) = myInnerProduct(A(index,:),b);
    end
end
```